Lab Assignment 2:
Recursion – Maze Solver

Due Dates	
Wednesday Labs	**Wednesday 20th Feb. @ 11:59**
Thursday Lab	**Thursday 21st Feb. @ 11:59**

Objectives

- Review manipulating two-dimensional arrays
- Review using basic classes and objects
- Review using recursion to solve a problem
- Follow the Software Life Cycle model, with pseudocode and code refinement steps

Problem Specification

Following the Software Life Cycle model presented in class, develop a Java application that implements a maze solver that will run on a robot. This maze solver randomly generates a maze (a two-dimensional **String** array) and attempts to find a path from the starting point in the maze to the final destination.

A Robot is to be programmed to navigate a maze and provide the result of that navigation to its master. The robot is required to start at the starting point (S) and find a path, if it exists, to the destination (D). Each location in the maze is either open or blocked (#). If there is at least one path from S to D, the robot should inform its master accordingly and show a map, or otherwise just tell the master no path exists from S to D.

The application should exhibit the following functionality (see the sample output below):

1. Generate a maze (a two-dimensional array).
    a. The numbers of rows and columns are given by the user, and must be in the range of [5, 10] (note: 5 and 10 are included), without the number of rows being the same as that of columns. For example, if the number of rows is 6, then the number of column cannot be 6.
    b. The program needs to validate input; if a particular input is invalid, the program should repeatedly request for that input (instead of terminating). Also, the program should print out corresponding reasons about why the input is invalid. This is demonstrated in the example output below. Once the number of rows and number of columns have been obtained, they will be used to generate the maze. The maze consists of four symbols as shown below:

- a number in the range [1, 100] (e.g., 5). All cells with a number indicate positions where the robot is able to pass through. The number also indicates the amount of coins that the Robot can collect at that position.
- "#" represents a blocked position. The robot cannot pass through this cell.
- "S" is the starting point for the robot and is at the top left corner of the maze.
- "D" is the robot's final destination – the end of the maze - and it is at the bottom right corner of the maze.

2. Randomly fill the maze with numbers in the range [1, 100] and #'s.
    a. The total number of #'s cannot be greater than a third of the total number of positions in the maze. For example, for a 4x5 maze, the total number of "#" cannot exceed 6.
3. Print out a maze to the screen once it is generated.
4. Attempt to find an available path from S to D.
    a. The robot is able to move one step every time in one of four directions (up, down, left and right – no diagonal movements) as long as there is no blocked cell in the specific direction.
        i. The program **MUST USE RECURSION** to find the path from S to D.
5. If the robot is not able to find a path, print a corresponding message to the screen.
6. If the robot finds a path, then:
    a. print out the maze with the path marked by +'s.
    b. sum up all the numbers on this path and print out the result as the amount of coins found.
7. The program is also required to run steps 2 - 6 above **3 times**. Each run of steps 2 - 6 should be labelled accordingly as shown in the sample output.

Your output must follow precisely the format of the Example Output below.

Example Output:

```
Enter number of rows in range [5, 10]:
1
Invalid input!
Enter number of rows in range [5, 10]:
a
Input must be an integer. Re-enter:
5
Enter number of columns in range [5, 10].
This must be different from number of rows:
5
Number of columns is the same as number of rows!
Enter number of columns in range [5, 10].
This must be different from number of rows:
6

******Maze #1******
Start drawing the maze...
The maze is as below:
S     #     6     #     4     100
27    78    #     #     51    #
27    #     #     #     3     #
```

```
47    29    47    #     53    #
32    29    48    56    47    D
Congratulations! I found a solution for this maze as below:
S     #     6     #     4     100
+     78    #     #     51    #
+     #     #     #     3     #
+     29    47    #     53    #
+     +     +     +     +     D
The amount of coins collected: 313

******Maze #2******
Start drawing the maze...
The maze is as below:
S     #     #     #     86    47
83    #     22    #     34    #
#     24    48    #     #     17
#     31    #     24    67    1
21    77    99    87    95    D
Sorry, no solution can be found for this maze!

******Maze #3******
Start drawing the maze...
The maze is as below:
S     #     57    71    5     37
88    #     #     72    #     #
#     68    79    #     #     #
88    69    47    #     #     #
55    35    89    99    26    D
Sorry, no solution can be found for this maze!
```

Design Requirements

You should have at least three classes in your application: **LA2Main, MazeInput** and **MazeSolver**. The basic structure (i.e. methods) of these classes have been provided for you in this assignment. However, you may have other attributes and/or methods **in addition to** those listed below:

LA2Main:

/*

The **main method** should be implemented to

- create instances of classes MazeInput and MazeSolver,
- call proper methods in MazeInput class to initialize MazeSolver,
- call proper methods in MazeSolver to start the maze.

This whole process needs to repeat three times, with different mazes generated each time.

*/

MazeInput:

/*
This method initializes/sets the numRows and numCols of the solver (instance of MazeSolver Class) with the numRows and numCols provided by the user.
*/

1. **public void initializeMazeSolver(MazeSolver solver);**

/*
This method asks for a number within the range [5, 10] from the user as the number of rows. The method is also responsible for validating the input and printing corresponding messages to the screen if the input is not valid.
*/

2. **private int getNumRows();**

/*
This method asks for a number within the range [5, 10] from the user as the number of rows - this number **cannot** be the same as the number of rows denoted by parameter numRows. This method is also responsible for validating the input and printing corresponding messages to the screen if the input is not valid.
*/

3. **private int getNumCols(int numRows);**

MazeSolver:

Hint: You need a boolean array with the same numbers of rows and columns as the maze to mark each position as true (visited or on the path of the solution) or false (unvisited or not on the path of the solution): **boolean visited[numRows][numCols];**

/* The setter for the private attribute: numRows. */

1. **public void setNumRows(int numRows);**

/* The setter for the private attribute: numCols. */

2. **public void setNumCols(int numCols);**

/* The getter for the private attribute: numRows. */

3. **public int getNumRows();**

/* The getter for the private attribute: numCols. */

4. **public int getNumCols();**

/*
Draw the maze (a 2-d matrix, type String) by randomly filling it with "#" (block) and numbers (e.g., "45"). Set the top left cell to "S" (starting point), and the bottom right cell to "D" (destination). Numbers are randomly generated within the range [1, 100], while the total number of "#" cannot exceed 1/3 of the total number of cells in the matrix.
*/

5. **public void drawMaze();**

   /* Print out the original maze. Each column must be properly aligned. */
6. **public void printMaze();**

   /*
   This method should only call the method "solveMazeRecursively(int row, int col)" – the recursive method - and return the result received from it.
   */
7. **public boolean isSolvable();**

   /*
   This method is the core part to solve the maze. It tries to solve the maze using recursion. It returns true if there is at least one solution. Otherwise, it returns false.
   */
8. **private boolean solveMazeRecursively(int row, int col);**

   /*
   If there is a solution, this method will be called to print out the maze with the path position replaced by "+". The amount of coins collected on the path is also calculated and printed in this method. Each column must be properly aligned.
   */
9. **public void printResult();**

NOTE: *Using recursion is one of the main objectives of this assignment. A solution that does not use recursion to search for the path will earn zero points.*

Hints

1. To generate the maze, you could first determine randomly if the current cell should contain a number or "#". If the cell is to contain a number, generate that number randomly. If the cell should contain "#", first check that the current number of blocks (i.e. "#") will not exceed a third of the number of cells in the maze. If that number will be exceeded, then you will need to generate a random number instead.

2. You may find it helpful to use **two** identical 2-dimensional arrays– one to represent the maze, and another one (of type **boolean**) that keeps track of which cells have been visited already or are part of the current path being discovered by the robot.

3. For the recursion, think about how you might check for a path if you were doing it by hand. You start from position [0][0] and check each of its neighbors (left, right, up and down) for a number. If one of the neighboring cells contains a number, you check its neighbors, and so on, until either reaching the end of the maze (D") or finding that the path is blocked. Think about which part of this is the recursive step, and what is / are the terminating condition(s).

Additional Requirements

Software Life Cycle Report

You are required to follow the Software Life Cycle (SLC) methodology presented in class, and write the SLC Report, explaining how you followed the nine phases of the Software Life Cycle in this assignment.

A proper *design* (with stepwise pseudocode refinement), a proper *coding* method (with stepwise code refinement starting from the most detailed pseudocode refinement), and proper *testing* are all essential. For reference, please see the **Sample SLC Report** (covered in class) on Elearning.

Note: The SLC report with correct pseudocode development will be worth 40% of the total LA grade.

You will need to **generate Javadoc** for your project. If Javadoc is correctly generated, a "**doc**" folder (for Javadoc) should be created in your project.

Coding Standards

You must adhere to all conventions in the CS 1120 Java coding standards (available on Elearning for your Lab). This includes the use of white spaces and indentations for readability, and the use of comments to explain the meaning of various methods and attributes. Be sure to follow the conventions also for naming classes, variables, method parameters and methods.

Assignment Submission

- Generate a .zip file that contains all your files including:
  - Program Files
  - Any input or output files
  - The SLC Report (a text with description of all nine phases of the Software Life Cycle)

- Submit the .zip file to the appropriate folder on ELearning.

NOTE: The eLearning folder for LA submission will remain open beyond the due date but will indicate how many days late an assignment was submitted where applicable. The dropbox will be inaccessible seven days after the due date by which time no more credit can be received for the assignment.

The penalty for late submissions as stated in the course syllabus will be applied in grading any assignment submitted late.